

2025年におけるChrome拡張E2Eテストのベストプラクティス

E2Eテストフレームワークの比較と推奨

Chrome拡張のE2Eテストには主に **Playwright**, **Puppeteer**, **Selenium(WebDriver)** の3つが候補に挙げられます。それぞれ特徴がありますが、2025年現在では **Playwrightが最も強力で信頼性が高い選択肢** と評価されています¹。以下では各ツールの特徴とChrome拡張テストへの適性を比較し、推奨手法を述べます。

- **Playwright** (Microsoft製) : Chromium (Chrome) だけでなくFirefoxやWebKit(Safari)にも対応した最新のE2Eフレームワークです。シングルAPIで複数ブラウザを操作でき、自動待機や並列実行、ネットワーク傍受など高度な機能を備えています²³。Chrome拡張のテストにも公式に対応しており、**拡張機能のUI (ポップアップやオプションページ) 、DOM変更、さらにはコンテンツスクリプトとバックグラウンド間のメッセージングの動作まで自動化可能**です⁴。Playwrightは専用のテストランナー (@playwright/test) を提供し、アサーションやレポートも含めオールインワンであるため、テスト開発の効率が非常に高いです。その**モダンな設計と安定性**から、現在多くのチームがPlaywrightを採用しています¹。
- **Puppeteer** (Google製) : Chrome/Chromium専用の自動化ライブラリで、Chrome DevTools Protocolを直接操作するため**Chromeブラウザに深く統合**され高い信頼性とパフォーマンスがあります⁵⁶。Node.jsで利用し、Playwrightほど高度なテストランナー機能は内蔵していませんが、任意のテストフレームワーク (MochaやJest等) と組み合わせてE2Eテストを構築できます。**Chrome拡張に限定してテストする場合、Puppeteerは軽量で有用**ですが³、ブラウザはChrome系のみとなり、多ブラウザ対応や高度な自動待機機能はPlaywrightに劣ります⁷。拡張のポップアップなども、後述する方法 (直接 `chrome-extension://` URLにアクセスする等) で**通常のページ同様に操作可能**ですが、コードは手動で多く書く必要があります。
- **Selenium WebDriver**: 最も歴史のある汎用E2Eテストフレームワークで、多言語対応や広範なブラウザサポートがあります。ただし**専用ブラウザドライバー (ChromeDriver等) を要する複雑なアーキテクチャ**であり、実行速度やセットアップの煩雑さで他の新しいツールに劣ります⁸⁹。Chrome拡張をテストする場合、ChromeDriverに拡張を読み込ませる設定は可能ですが、**拡張UIの操作やコンテンツスクリプトとの相互作用を制御するのは難易度が高い**です。専門知識が必要で、加えてSeleniumは自動待機の調整や安定化に手間がかかるため¹⁰、コストパフォーマンスの点で他の選択肢より劣るでしょう。

以上を踏まえると、**最も推奨されるのはPlaywrightを用いたE2Eテスト**です。Playwrightなら**Chrome拡張のポップアップからコンテンツスクリプトまで含めて、エンドユーザー操作を近い形で再現したテスト**が可能です⁴。PuppeteerもChrome限定で軽量の代替として有用ですが、将来的な保守性・拡張性を考えるとPlaywrightの方が優れています³¹。Seleniumはレガシーなケースや既存資産がある場合を除き、Chrome拡張E2Eにはあまり向きません。

Chrome拡張E2Eテストのポイント (UI操作とメッセージング)

Chrome拡張固有のテスト上の課題として、**複数の実行コンテキストと特別なUI要素**があります¹¹¹²。例えば「拡張ポップアップ (ブラウザツールバーのアイコンをクリックすると出るUI)」 「コンテンツスクリプ

ト (任意のWebサイトに注入されるスクリプト)」「オプションページ」「バックグラウンドスクリプト (サービスワーカー)」などがそれぞれ独立した環境で動作します¹¹。E2Eテストではこれらが連携してユーザー体験を形成する一連の流れを検証する必要があります。

Playwrightを用いた場合、この課題に対して以下のようなアプローチが有効です。

- **拡張機能の読み込み:** テスト用ブラウザを起動する際に、拡張をパス指定で読み込みます。Playwrightなら `chromium.launchPersistentContext(userDataDir, { headless: false, args: [...] })` で `--disable-extensions-except` と `--load-extension` フラグにビルド済み拡張のパスを渡すことで、拡張を有効化したChromiumコンテキストを作成できます^{13 14}。このとき**headlessモードは使用不可**なので `headless: false` にします (Chromeの拡張はヘッドレスでは動作しません)^{15 16}。
- **動的な拡張ID取得:** Chrome拡張の内部ページ (ポップアップやオプションページ) のURLにはランダムな拡張IDが含まれます (`chrome-extension://<EXTENSION_ID>/popup.html` 等)¹⁷。環境ごとに変わるこのIDをテスト中に取得するため、Playwrightで `chrome://extensions/` にナビゲートして拡張IDを読み取る方法があります¹⁸。具体的には、`chrome://extensions` を開きデベロッパーモードを有効化した上で拡張カード要素からID属性を取得します¹⁸。こうして**テスト実行時に拡張IDを動的に取得し、以降の操作に利用**することで、ハードコーディングを避けて汎用的なテストを実現できます^{18 19}。
- **ポップアップUIの操作:** ブラウザツールバーから直接ポップアップを開く操作は自動化が困難ですが、代替策として**ポップアップページを新規タブで直接開く**ことが可能です²⁰。取得した拡張IDを用いて、例えば `await context.newPage().goto("chrome-extension://<ID>/popup/index.html")` のようにナビゲートすると、**ポップアップと同じUIを通常のページとして操作**できます²¹。これは実際のユーザー操作 (アイコンクリック) とは異なりますが、内部的な動作検証には十分であり、ボタンクリック等の操作も通常のページ同様に可能です^{21 22}。Playwrightであればこのポップアップ用のPageオブジェクトと、任意のWebサイト上で動作するコンテンツスクリプト用のPageを同時に扱えます。
- **メッセージングとコンテンツスクリプトの検証:** 拡張ポップアップからコンテンツスクリプトへメッセージを送りDOMを書き換えるような挙動も、E2Eテストで再現・検証できます。Playwrightでは**複数のページ (タブ) を開き、それぞれを操作して相互作用をテスト**できます^{23 24}。例えば:
 - **テスト対象サイトを開く:** `page.goto("https://example.com")` 等で通常のコンテンツページを開きます (ここに拡張のコンテンツスクリプトが挿入される)²⁵。
 - **拡張ポップアップを開く:** 上述の方法で `popupPage` を新たなタブとして開きます²⁵。
 - **フォーカスの制御:** ポップアップを開いた直後はそのタブがアクティブですが、例えば「ポップアップで設定した内容が元のページに反映される」ケースでは元のコンテンツページをアクティブに戻す必要があります^{26 27}。Chrome拡張はポップアップを開いている間バックグラウンドでコンテンツページを更新するため、Playwrightの**Chrome DevTools Protocol (CDP)機能**を利用して `Page.bringToFront` で任意のタブを前面に復帰させることができます²⁸。上記例でも、CDP経由でコンテンツページをアクティブに戻してからポップアップ内のボタンをクリックし、その後コンテンツページ側で期待する要素が表示されるか検証しています^{29 27}。
 - **結果の検証:** 最後にコンテンツページ上で、拡張が注入したDOM (例ではタイマー表示要素) が正しく表示されているか `page.waitForSelector` や `expect` アサーションで確認します²⁷。

以上のように、Playwrightなら**拡張ポップアップ～コンテンツスクリプト間のシナリオもエンドツーエンドで自動テスト可能**です^{25 27}。Puppeteerでも同様の手法 (Chrome拡張のURLに直接アクセス、CDPの利用) で実現はできますが、これらの制御ロジックを自前で実装する必要があります。一方Playwrightはこれら機能をサポートする高水準APIや豊富なドキュメントがあり、**テストコードの可読性・保守性に優れる**点で

「コストパフォーマンス」が高いと言えます⁴³⁰。したがって、Chrome拡張のUI操作やメッセージング検証にはPlaywrightを用いたE2Eテストを第一候補として推奨します。

GitHub ActionsでのE2Eテスト実行方法

CI環境（特にGitHub Actions）上でChrome拡張のE2Eテストを実行する際には、**GUIブラウザを起動する必要がある**点に注意が必要です。Chrome拡張は前述の通りヘッドレスモードでは動作しないため、CI上でもヘッドフル（画面表示あり）でブラウザを起動します¹⁶。GitHub Actionsのデフォルトランナー（Ubuntuベース）では画面がないため、**仮想ディスプレイ (Xvfb)** を使ってブラウザを動作させます¹⁶。

具体的な手順は以下の通りです。

- 1. テスト環境セットアップ:** `actions/checkout@v3` でリポジトリをチェックアウトし、`actions/setup-node@v3` 等でNode.jsをセットアップします。その後、依存関係をインストールし、Playwrightの場合は追加でブラウザのインストールも行います。例として、Railware社のブログではPlaywright導入時にCI用にブラウザをインストールするWorkflowが紹介されています³¹（後述のキャッシュ節も参照）。
- 2. Xvfbによるヘッドフル実行:** GitHub Actions上でヘッドフルモードのブラウザを起動するには、コマンド実行前に`xvfb`を起動します。簡便な方法は、テスト実行コマンドを`xvfb-run`で包むことです。例えばPlaywrightの場合、通常ローカルでは`npx playwright test`と実行しますが、CIでは次のようにします³²：

```
- name: Run Playwright tests (headed)
  run: xvfb-run npx playwright test
```

これにより仮想ディスプレイ上でブラウザGUIを起動し、拡張機能付きのテストを実行できます³³。PuppeteerやSeleniumでも同様に、テスト起動を`xvfb-run`でラップするか、またはジョブ内で`Xvfb :99 &`を起動してDISPLAY環境変数を設定する方法があります。

- 3. GitHub Actions上でのブラウザと依存関係:** Playwrightの場合、公式ドキュメントにCI環境用のセットアップガイドがあります³³。`npx playwright install-deps`を使用すると、ブラウザ実行に必要なOSパッケージ（フォントやライブラリ類）をUbuntuにインストールできます³⁴。先に`npx playwright install --with-deps`を実行してPlaywrightのChromiumブラウザを取得しておけば、`install-deps`で必要パッケージを入れるという流れです³¹。Puppeteerの場合も、UbuntuランナーにはGoogle Chromeがインストール済みであるため（多くのCI環境ではChrome/Chromiumが用意されています）、必要に応じて`apt install`で不足ライブラリを入れたり`--no-sandbox`オプションを付けたりします。また、Seleniumを使う場合は`services: [selenium]`や`browser: chrome`といった設定でChromeDriverを起動できますが、拡張を読み込むにはChromeDriverのオプションで拡張パスを指定する必要があります。
- 4. ワークフローファイル:** Playwrightを導入する際に自動生成されるGitHub Actions用テンプレート（`.github/workflows/playwright.yml`）をベースにするのが簡単です³⁵。その場合も上記のように`xvfb-run`の追加だけ注意してください³³。自動生成されない場合でも、ジョブでUbuntuランナーを使用し、上記ステップを順番に記述すれば問題ありません。
- 5. Artefactアップロード（任意）:** テスト結果（動画やスクリーンショット、レポート）を保存したい場合、`actions/upload-artifact@v3`で保存するとデバッグに役立ちます³⁶。

要点として、GitHub Actions上でChrome拡張のE2Eテストを実行することは可能ですが、「ヘッドレス不可」「Xvfb利用」の2点が通常のWebアプリテストと異なります¹⁶。Playwright+GitHub Actionsの組み合わせなら公式にもノウハウが蓄積されており、比較的容易にCIに組み込めるでしょう。

ブラウザのダウンロード時間を最小化するテクニック

CI実行時の待ち時間を短縮するために、ブラウザ（特にChrome/Chromium）のインストール・ダウンロード時間を減らす工夫が重要です。以下にコストパフォーマンスの高い手法を挙げます。

- **不要なブラウザをインストールしない:** PlaywrightはデフォルトでChromium・Firefox・WebKitの各ブラウザをダウンロードしますが、Chrome拡張のテスト用途であれば**Chromiumだけで十分**です。初期設定の段階で他ブラウザを無効にすると、ダウンロード時間と容量を節約できます。例えばPlaywrightの設定ファイル `playwright.config.ts` でプロジェクトをChromiumのみに限定したり³⁷、Playwrightのインストール時に対話オプションで「GitHub Actions用ワークフローを追加」→「ブラウザをインストール」でChromium以外をスキップする、といった対応が可能です³⁸。Railswareのブログでも、Chrome拡張テストのためにFirefox/WebKitを設定から外すことを推奨しています³⁷。
- **キャッシュの活用:** ブラウザのバイナリや依存ファイルをキャッシュすることで、毎回のジョブで再ダウンロードする必要を無くせます。GitHub Actionsでは `actions/cache` を利用し、Playwrightの場合は既定のキャッシュディレクトリ（Linuxでは `~/.cache/ms-playwright`）をキー付きで保存できます³⁹。例えば以下のように設定し、一度ダウンロードしたPlaywrightブラウザを以降のジョブで再利用できます³⁹：

```
- name: Cache Playwright browsers
  uses: actions/cache@v3
  with:
    path: ~/.cache/ms-playwright
    key: ${{ runner.os }}-playwright-${{ env.PLAYWRIGHT_VERSION }}
```

こうすることで、Playwrightのブラウザ更新がない限りキャッシュから高速に展開でき、約10秒程度とされるダウンロード時間を省略できます⁴⁰。（※MicrosoftはPlaywrightブラウザのキャッシュは必須ではないとコメントしていますが、CI高速化には有効です。）

Puppeteerを使う場合も、`node_modules` 内に含まれるブラウザをキャッシュするか、**そもそもダウンロードをスキップ**することが可能です。環境変数 `PUPPETEER_SKIP_CHROMIUM_DOWNLOAD=true` を設定してPuppeteerをインストールすると、Chromiumの自動取得を省略できます⁴¹。その代わりにGitHub Actionsランナーにインストール済みのChromeを使うようPuppeteerに指定します（例えば `puppeteer.launch({executablePath: '/usr/bin/google-chrome-stable'})`）。Ubuntuランナーには通常Google Chrome Stableがプリインストールされているため、この方法でダウンロード時間をゼロにできます。

- **プリビルドされたブラウザの利用:** ブラウザのダウンロードを避けるもう一つの手段として、**事前にブラウザを含んだコンテナやアクションを使う**方法があります。例えばPlaywright公式のDockerイメージ（`mcr.microsoft.com/playwright:v<version>`）にはChromium等が含まれており、`jobs.<job>.container` オプションでそのイメージを指定してジョブを実行すれば、ブラウザDLを省略できます⁴²。もっとも、Dockerイメージ自体のプルに時間がかかったり、環境管理が複雑になる可能性もあるため、シンプルなキャッシュ戦略で足りている場合は無理に使う必要はありません。GitHub Actions上で利用可能な**公式セットアップアクション**も活用できます。例えば `browser-`

`actions/setup-chrome`のようなコミュニティアクションがあれば、簡単に最新Chromeをインストールできますが、Ubuntuランナーには既にChromeがありますので大半のケースでは不要でしょう。

以上のテクニックにより、CIでのブラウザ取得コストを大幅に削減できます。特に**キャッシュキーにブラウザバージョンを入れる**ことで、ブラウザ更新時のみ再ダウンロードし、それ以外は高速に進行させる運用が現実的です³⁹。Chrome拡張のE2EテストではChrome/Chromium以外のブラウザは基本不要ですので、その点を割り切った最適化が有効です。

GitHub Actionsにおける安定性・実行時間最適化の設定

最後に、GitHub Actions上でChrome拡張のE2Eテストを安定かつ効率的に実行するための設定ポイントをまとめます。

- **ランナー環境 (`runs-on`) の選択:** 基本的には `ubuntu-latest` (Ubuntu 22.04 など) が推奨されます。Ubuntuランナーはセットアップ時間が短く、多くの必要パッケージがあらかじめ揃っており、Chromeブラウザもインストール済みです。WindowsやmacOSランナーでも実行は可能ですが、起動に時間がかかったりGUI周りで余計な調整が必要になることがあります。したがって**特別な理由がなければUbuntuランナーを使用**するのがコスト的にもベストです。
- **依存関係のキャッシュ:** 上述のブラウザキャッシュに加え、`node_modules`などの依存も `actions/cache` で保存するとよいでしょう。特にPlaywright含むE2Eテスト関連のライブラリは容量が大きいので、パッケージの再インストール時間を削減することでCI全体の時間短縮につながります。キャッシュキーには `package-lock.json` や `pnpm-lock.yaml` のハッシュを用いることで、依存が変わったときのみ更新されるようにします。
- **テスト並列化:** テストケースが多数ある場合、**並列実行による時間短縮**が可能です。Playwrightのテストランナー(`@playwright/test`)はデフォルトでマルチスレッド実行を行いますが、GitHub Actions上の2コアマシンではデフォルト2並列程度になります。もっと高速化したい場合は、**ジョブを分割 (シャーディング)** して並列に走らせることも検討できます⁴³。例えばPlaywrightはビルトインで**テストのシャーディング機能**があり、`npx playwright test --shard=1/2` のように全テストをグループ分けできます⁴³。GitHub Actionsの `matrix` 戦略と組み合わせると `shard 1/2` と `2/2` を同時実行すれば、テスト時間を半減できる可能性があります⁴⁴。ただし、拡張機能のテストはブラウザ実行が重いので、並列度を上げすぎるとリソース不足や不安定さに繋がる点には注意が必要です。
- **リソースとタイムアウト設定:** 安定性のためには、タイムアウトやリトライの設定も重要です。GitHub Actionsのジョブに `timeout-minutes` を設定して無限実行を防いだり⁴⁵、Playwright側で各テストに適切なタイムアウトを指定しておきます。ネットワーク遅延など環境差で一時的なテスト失敗が起きる場合、Playwrightの `retry` オプションで自動リトライを設定するとCIの信頼性が上がります。拡張機能テストではポップアップ表示やメッセージングに実時間がかかることもあるため、ローカル環境よりも**余裕を持った待ち時間**を設定しておくとうまいでしょう。
- **CIマシンスペックの考慮:** GitHubのホストランナーは無料枠では限られた性能です。テストが重く実行時間が長い場合、`ubuntu-latest` でも複数ジョブが重なると遅延する可能性があります。もし予算や必要性があれば、`runs-on: ubuntu-20.04-4core` といった **大きめのマシンタイプ** や self-hosted ランナーで実行する選択もあります。ただし通常のChrome拡張テスト規模であれば、2コアでも十分実用的でしょう。
- **他のベストプラクティス:** 不要なログ出力を抑制する、Artifactsを必要な場合のみアップロードする、なども全体のスピードに影響します。加えて、CI実行順序としてPull Request時のみフルE2Eを回し、push時は簡易なチェックに留めるなど運用レベルでの工夫も有効です。

以上の設定により、GitHub Actions上でのChrome拡張E2Eテストは**安定かつ可能な限り短時間**で完了するよう最適化できます。特に**Xvfbを用いたヘッドフル実行**というポイントさえ押さえれば、あとの部分（キャッシュや並列化）は通常のWebアプリE2Eと同様のベストプラクティスが適用できます。 33 31

まとめ

本ガイドでは、2025年現在におけるChrome拡張のE2Eテスト手法について、ツール選定からCI実行まで包括的に解説しました。結論として、**Playwrightを用いたアプローチが最も有力**であり、拡張特有のポップアップやコンテンツスクリプト間連携も含めたテストを実現できます 4。GitHub Actions上でもXvfbによるヘッドフル実行でこれらテストを自動化でき、キャッシュ活用や並列実行によってCI時間を短縮しつつ安定性も確保可能です。

Chrome拡張の品質を高めるため、E2Eテストは有効な手段です。初期設定に多少工夫は要りますが、一度整えれば通常のWebアプリと同様に継続的な自動テストが可能になります。ぜひ本記事の手法を参考に、**Dropbox動画プレイヤー操作やツールチップ挿入といったシナリオ**を含めた包括的なテスト体制を構築してみてください。適切なフレームワークとCI最適化により、コストパフォーマンス良く拡張機能の信頼性を担保できるはずです。 1 30

1 5 6 7 8 9 10 Choosing between Playwright, Puppeteer, or Selenium? We recommend Playwright

<https://www.browserbase.com/blog/recommending-playwright>

2 3 End-to-End Testing in 2025: Complete Beginner's Guide

<https://www.bunnysHELL.com/blog/introduction-to-end-to-end-testing-everything-you-/>

4 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 How I Built E2E Tests for Chrome Extensions Using Playwright and CDP - DEV Community

<https://dev.to/corrupt952/how-i-built-e2e-tests-for-chrome-extensions-using-playwright-and-cdp-11fl>

31 34 36 39 45 How To Cache Playwright Browser On Github Actions - DEV Community

<https://dev.to/ayomiku222/how-to-cache-playwright-browser-on-github-actions-51o6>

32 33 35 37 38 How to Test Chrome Extensions with Playwright | Railsware Blog

<https://railsware.com/blog/test-chrome-extensions/>

40 How to cache on github actions? · Issue #7249 · microsoft/playwright

<https://github.com/microsoft/playwright/issues/7249>

41 Puppeteer is not able to install: "ERROR: Failed to set up Chromium ...

<https://stackoverflow.com/questions/63187371/puppeteer-is-not-able-to-install-error-failed-to-set-up-chromium-r782078-set>

42 Continuous Integration - Playwright

<https://playwright.dev/docs/ci>

43 Sharding | Playwright

<https://playwright.dev/docs/test-sharding>

44 Running Playwright Tests in Parallel: Speed Up Your Test Suite

<https://medium.com/@testrig/running-playwright-tests-in-parallel-speed-up-your-test-suite-a33c86f6e512>